SAND2022-2691C

Exploring Spatial Indexing for Accelerated Feature Retrieval in HPC

Margaret Lawson
Sandia National Laboratories &
Univ. of Illinois at Urbana-Champaign
mlawso@sandia.gov

William Gropp
Univ. of Illinois at Urbana-Champaign
wgropp@illinois.edu

Jay Lofstead
Sandia National Laboratories
gflofst@sandia.gov

***Abstract*—Despite the critical role that range queries play in analysis and visualization for HPC applications, there has been no comprehensive analysis of indices that are designed to accelerate range queries and the extent to which they are viable in HPC. In this paper we present the first such evaluation, examining 20 open-source C and C++ libraries that support range queries. Contributions of this paper include answering the following questions: which of the implementations are viable in HPC, how do these libraries compare in terms of build time, query time, memory usage, and scalability, what are other trade-offs between these implementations, is there a single overall best solution, and when does a brute force solution offer the best performance? We also share key insights learned during this process that can assist both HPC application scientists and spatial index developers. While we find that there is no single best solution, three libraries, Boost, CGAL and R-tree, offer some of the best performance, scalability, memory overheads, and support for different mesh types. We find several areas where the spatial indices could be substantially improved: better performance when there are a large number of query matches, reduced memory overheads, and better support for GPUs or other accelerators.**

***Index Terms*—geometric range searching, spatial indexing, k-d tree, R-tree, octree**

I. Introduction

During analysis and visualization, HPC application scientists often need to extract particular spatial subsets of their data. Scientists may need to extract these subsets for a number of reasons including sub-sampling [1], to perform areal interpolation [2], [3], or to perform spatial correlations [4]. Scientists may also extract spatial subsets that correspond to regions or features of interest. This can allow scientists to concentrate their analysis and visualization on more interesting data or to use different analysis routines depending on the type of region. Examples of regions that may be extracted include application boundaries and surfaces [5], regions of the brain in neuroscience imagery [6], counties or other regional boundaries in satellite imagery [7], regions with different flow types in a fluid dynamics simulation [8], and regions with different chemical properties in a combustion simulation [9]. Examples of spatial features of interest that may be extracted include supernovae in astronomy [10], blood vessels, lesions, and areas of inflammation in pathology imagery [11], shocks in a fluid simulation [7], flame fronts, extinction events and vortices in a combustion simulation [9], and tropical storms in a climate simulation [12]. Extracting spatial subsets is thus a critical component of many different types of analysis and visualization routines for a wide range of HPC applications.

Extracting a particular spatial subset is trivial for regular meshes (meshes in which the cells are congruent parallelotopes) where a single, simple function can be used to map from a spatial coordinate to a data array offset. In the simplest case (a Cartesian mesh), the mesh coordinates will be the same as the array indices. In contrast, for unstructured meshes (meshes without implied neighborhood connectivity), this operation is very computationally intensive. Without any form of index, scientists will have to perform a linear (brute force) search over all mesh points or elements for each region they want to extract. With petascale simulations already using meshes that can range from tens of billions [13], [14] to trillions [15] of elements and with the impending arrival of exascale machines [16], this kind of linear search, which may need to be performed thousands or millions of times during analysis and visualization, typically represents a significant inefficiency that can delay the path to scientific discovery. A lot of research has been done to develop spatial indices that can accelerate this type of search (known as a range query) to help scientists quickly identify what mesh points or elements fall within a given region. However, there has been no comprehensive analysis of spatial index implementations that are designed to accelerate range queries and the extent to which they are viable in an HPC setting. To be viable in an HPC setting, a spatial indexing library must have good performance for building the index and performing queries, good scalability, and moderate memory overheads. It is important for the index to have moderate memory overheads since the index will provide the best performance when stored in memory, many applications require significant amounts of memory (for MPI, for the mesh, for variable data, etc.) and nodes have limited amounts of memory.

In this paper, we present the first comprehensive review of spatial index implementations that support range queries. We also provide scientists with the information they need to decide when to use a spatial index for range querying and which spatial index implementation will be best suited to their use case. This information can help scientists accelerate range queries by several orders of magnitude and thereby greatly accelerate their analysis routines and the discovery process. Contributions of this paper include presenting a thorough analysis of 20 free, open-source C and C++ libraries that support range queries with

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

expected sub-linear query times, sharing key insights learned during this process, and answering the following questions:

1) Which of the implementations are viable in HPC?
2) How do these libraries compare in terms of build time, query time, and memory usage at different scales?
3) Is there a single overall best solution?
4) When should a brute force solution be used?

The rest of this paper is organized as follows. Section II discusses related work. Section III provides an overview of the evaluated libraries. Section IV presents the evaluation and results. Finally, Section V provides discussion, offers additional insights, and presents areas for future work.

II. Related Work

Range queries have been extensively studied in many different branches of computer science including privacy and security, mobile computing, networks, cloud computing, and databases. Below we provide an overview of two categories of related work that are most related to the work presented here: theoretical research done on geometric range searching and search structures and research that performs a comparison of several different spatial indexing libraries.

A. Theory: Geometric Range Searching and Search Structures

Efficient geometric range searching has been the focus of a significant amount of research in the theoretical branch of computer science. A good overview of this work can be found in [17], [18], [19], [20]. This work has invented a number of geometric search structures, and here we present an overview of the three that are most commonly supported by the libraries we evaluate in this paper: octrees, k-d trees, and R-trees.

1) Octrees: An octree [21] is a tree in which each internal node has exactly eight children and which is not necessarily balanced. An octree can either be used to recursively partition a 3D space into eight octants (region-based octrees) or to partition a set of 3D points based on their coordinates (point-based octrees). Internally, octrees typically store point data only in the leaf nodes for region-based octrees and and in both the leaf nodes and internal nodes for point-based octrees [22].

2) K-d trees: K-d trees [23] are binary trees that store k-dimensional data and represent a recursive subdivision of this data using $(k-1)$-dimensional hyperplanes. Each (non-leaf) level in the tree corresponds to one of the k dimensions, and each internal node at that level represents a splitting hyperplane for that dimension. K-trees are sensitive to the order in which points are inserted and are not necessarily balanced. Often the data will be split before the median point along the longest side of the node (the sliding midpoint rule).

3) R-trees: R-trees [24] are balanced trees that contain a hierarchy of d-dimensional boxes, where each child node is contained in the box represented by its parent. Depending on the implementation, these partitions may overlap. The leaves either store a d-dimensional point or the d-dimensional minimum bounding box of the objects stored in the leaf. These objects may be a set of points or other shapes. R-trees have a minimum and maximum branching factor for all internal nodes (apart from the root), and there are algorithms designed to build optimal R-trees using bulk (static) construction [25], [26]. R*-trees are an R-tree variant that is designed to minimize coverage and spatial overlap between internal nodes.

B. Comparisons of Libraries Supporting Range Queries

Although many research efforts have implemented data structures that support range queries, almost no work has performed a comparative evaluation of several spatial indexing libraries. One project performed a comparison of nearest neighbor search implementations and included some evaluation of range queries [27]. However, this work only considers six of the libraries we evaluate here, collects range query results only for artificial datasets containing 60,000 points, and was performed almost a decade ago.

III. Spatial Indexing Libraries Overview

We evaluate 20 libraries that support range queries. To be included in this study, a library has to: provide bindings for C or C++, offer a free, open-source version, and offer spatial indexing that supports range querying (with expected sub-linear query times). We limit our search to libraries with C or C++ bindings since these are the languages most commonly used in HPC for analysis and visualization tasks [28], [29]. Only one library, LEDA [30], is excluded for failing to meet the free, open-source requirement. LEDA offers a free version, but range queries are only offered in its paid version, which costs 3000 euros for pure research efforts. We leave for future work evaluating libraries that offer other language bindings.

Table I contains basic information about the libraries. In the evaluation, we perform box queries since this is the type of query used most commonly in analysis and visualization. For libraries that do not support box queries, we use a sphere (or radius) that encompasses the box, and then perform a filtering step to ensure exact matches are returned. This will result in a slight performance penalty for these libraries.

Note that some of the libraries share the same name or similar names (e.g., KDTREE and KDTREE2, libkdtree++ and libkdtree). We have added a number (shown in parentheses) after some of these library names to help disambiguate them. For the rest of the paper, we will refer to these libraries using the name appended with the number (without the parentheses).

Insight: few libraries can store mesh elements (shapes/boxes). Since many variables are calculated per mesh element, this is a significant limitation.

For more information about these libraries, readers can look at the Git repository for this project [50].

IV. Evaluation and Results

This section presents an overview of the evaluation setup including the hardware and software, datasets, and scales used in testing. We also present the performance and scalability results. HPC application data is typically collected for each mesh node or mesh element. This is why we perform an evaluation for both mesh points and elements.

TABLE I: Basic info. about the libraries evaluated in this paper

Library	Ver.	Tree type	Search type	Data types	Leaf size
3DTK [31]	-	**k-d tree**	**Box**	**Points**	**20**
ALGLIB [32]	**3.17.0**	**k-d tree**	**Box**	**Points**	**8**
ANN [33]	**1.1.2**	**k-d tree**	**KNN w. radius**	**Points**	**200**
ANN [33]	**1.1.2**	**BD-tree**	**KNN w. radius**	**Points**	**200**
Boost [34]	**1.74.0**	**R-tree**	**Box**	**Points, boxes**	**200**
CGAL [35]	**5.2.0**	**k-d tree**	**Box**	**Points**	**200**
CGAL [35]	**5.2.0**	**Box**	**Box**	**Points**	**1**
CGAL [35]	**5.2.0**	**Segment**	**Box**	**Boxes**	**1**
CGAL [35]	**5.2.0**	**R-tree**	**Box**	**Points, shapes**	**1**
FLANN [36]	**1.9.1**	**k-d tree**	**Sphere**	**Points**	**200**
KDTREE(1) [37]	-	**k-d tree**	**Sphere**	**Points**	**1**
KDTREE2 [38]	-	**k-d tree**	**Sphere**	**Points**	**200**
KDTREE(3) [39]	-	**k-d tree**	**Sphere**	**Points**	**200**
KDTREE(4) [40]	**0.5.7**	**k-d tree**	**Sphere**	**Points**	**1**
libkdtree++ [41]	-	**k-d tree**	**Sphere**	**Points**	**1**
libkdtree(2) [42]	-	**k-d tree**	**Box**	**Points**	**1**
libnabo [27]	**1.0.7**	**k-d tree**	**KNN w. radius**	**Points**	**200**
libspatialindex [43]	**1.9.3**	**R*-tree**	**Box**	**Boxes**	**200**
nanoflann [44]	**1.3.2**	**k-d tree**	**Sphere**	**Points**	**200**
Octree [45]	-	**octree**	**Sphere**	**Points**	**200**
PCL [46]	**1.11.1**	**k-d tree**	**Sphere**	**Points**	**15**
PCL [46]	**1.11.1**	**octree**	**Box**	**Points**	**20**
PicoTree [47]	**0.5.2**	**k-d tree**	**Box**	**Points**	**20**
R-tree [48]	-	**R-tree**	**Box**	**Boxes**	**20**
Spatial [49]	**2.1.8**	**k-d tree**	**Box**	**Points, boxes**	**1**

A. Experimental Setup

All experiments are run on the Vortex machine at Sandia National Labs. Vortex uses RHEL7, and has the following per node: 318 GB high-bandwidth DRAM and dual socket IBM POWER9 CPUs with 22 cores/socket and 4 hardware threads per core (176 total). We use GCC 10.2.0 to compile all of the libraries apart from PCL, which we can only build with GCC 8.3.1 on Vortex. We use four different evaluation setups, which are described in Table II. All experiments use the same mesh, which was created for a typical [51] turbulent low-Mach study that includes solution verification. The turbulent low-Mach study uses meshes that range from around 150 million to 2 billion nodes and elements. The mesh we use is unstructured and has 152.7 million nodes and 152.1 million hexahedral elements. The coordinates are 8 byte doubles. It should be noted that there is no standard in HPC for the number of mesh points or elements assigned per process during domain decomposition. The number will depend on a number of factors such as the computational intensity of the application, the memory per node, and the number of compute hours the scientist has access to. However, using between $10,000$ and $100,000$ nodes or elements per process during a simulation is common [52], [53], [54], [55]. Then, during analysis and visualization, scientists will often use a much smaller number of processes (10% or fewer) [56], [57], [58] thus resulting in upwards of $100,000$ to $1,000,000$ mesh nodes or elements per process. This is further evidenced by the fact that analysis and visualization clusters often have far fewer cores and nodes than compute clusters [59], [60]. We therefore perform “small” scale evaluation with approximately $1,000,000$ mesh nodes or elements per process to reflect this common range, and “large” scale evaluation with approximately $10,000,000$ mesh nodes or elements per process to reflect more extreme use cases. We use OpenMPI 4.1.0 with one hardware thread per (MPI) process. Apart from the weak scaling results, all evaluations use a single node. But, as the weak scaling results demonstrate in Section IV-C2, these results should hold at any scale if the number of processes per node is held constant. This is because each process performs entirely independent work without any need for message passing or coordination, and the indices are also kept entirely in memory (eliminating PFS contention).

TABLE II: Experiment setups

	Mesh points	Mesh elements
160 procs. (small scale)	1,227,411 / proc.	1,087,807 / proc.
16 procs. (large scale)	10,163,37 / proc.	10,878,078 / proc.

B. Performance Evaluations

Each performance evaluation consists of two basic parts. First, each process creates an in-memory instance of the data structure being evaluated and inserts the mesh coordinates or elements for its subdomain (assigned portion of the simulation space). Second, each process performs a set of queries at four different sizes, which we refer to as extra-small, small, medium and large. These queries return approximately 0.001%, 0.1% 1%, and 10% of the data. Each process performs 10,000 queries that are extra-small, 10,000 that are small, 10,000 that are medium, and 1,000 that are large. This allows us to evaluate how the spatial indices perform for retrieving features or regions of interest that range from very small to very large. This testing workload is designed to reflect the real-world scenario described and cited in the introduction: when a scientist performing analysis or visualization with a complex mesh type needs to extract data for particular regions of interest. In the large scale evaluation, we evaluate all libraries that perform within a factor of $10\times$ of the best performing library for each of the query sizes. This helps us to identify which libraries perform “best” across different query sizes and at different scales. For all libraries that allow the user to set a maximum leaf size, we test leaf sizes of 20 and 200. For space reasons, we only present the best result for each library. The full set of results can be found in the Git repository for this project [50]. Several libraries also offer different insertion algorithms or splitting strategies (for k-d or BD trees). However, we did not find any significant performance differences between these algorithms and have therefore omitted these results to save space. One exception is that we use Boost’s STR packing algorithm for all of the Boost tests since it results in dramatically improved results for data that can be bulk loaded (like mesh data). For libraries that offer both static and dynamic trees, we use static trees since the mesh we use in evaluation is static. We perform three runs for each configuration and average the results. After completing the timing results, we also collect memory results using the Massif heap profiler [61] from Valgrind 3.16.1 [62]. For these memory results, we present data only on the construction of the tree since memory increases during querying are transient. We exclude memory allocated as a result of reading in the mesh nodes or elements from the mesh file. We run these tests on a single process that has the amount of data closest to the average (within 3% of the global average).

In addition to the libraries, we evaluate a brute force solution. In the build phase, the solution copies the 2D vector of mesh points or elements using the assignment operator. In reality, no such copy would be required, we merely include this result as a point of comparison. For the queries, the brute force solution performs a linear search checking for each point or element if it intersects the query range. Evaluating the brute force solution allows us to determine how much of a performance advantage the spatial indices offer and to evaluate whether this advantage is sufficient to justify the memory requirements of the index.

In each of the result tables below, for the build throughput (insertions per second), and the query throughput (queries per second), we classify a library's performance as fast (green cell with bolded text) if is within 10× of the best performing library for build time or for the given query size. We classify the performance as moderate (yellow with plain text) if it is within 100×. All other results are classified as slow (red with underlined text). For memory usage, we classify any configuration that uses 2× or less of the raw memory size as having low memory requirements (green with bolded text), any configuration that uses less than 5× the raw memory size as having moderate memory requirements (yellow with plain text), and all other configurations as having high memory requirements (red with underlined text). In all cases, the raw data size can be seen by looking at the memory requirements for the brute force solution, which makes a copy of the mesh points or elements. In the tables, memory (abbreviated mem.) indicates the memory used by the tree, while peak memory indicates the maximum memory used while creating the tree.

Insight: some libraries temporarily use large amounts of additional memory making them a poor choice for significantly memory constrained environments.

1) Small Scale Mesh Points: Each query of extra-small, small, medium, and large size retrieves an average of 0.00104%, 0.100%, 1.03% and 8.85% of the process's assigned mesh points. Even at this small scale, a few libraries run into errors and are thus not included in the results. First, although FLANN and PCL can use GPUs, with 4 NVIDIA Tesla V100 GPUs with a total of 64 GB RAM, they run into out of memory errors. In addition, since they are the only two libraries that support GPUs it is not possible to perform a true cross-library comparison of this capability. Next, the CGAL range tree experiences out of memory errors. Finally, none of the ANN configurations complete within 24 hours. We therefore include results only for the query sizes that complete in these 24 hours.

Insight: only two of the libraries can utilize GPUs and they quickly exhaust the GPUs memory.

TABLE III: Small scale results for mesh points. Some libraries do orders of magnitude better on writing, querying, memory usage[1].

Configuration	**Writes/sec (millions)**	**Queries/sec**				**Mem. (MB)**	**Peak Mem. (MB)**
		XS	**S**	**M**	**L**		
Brute force	**5.51**	29.3	29.3	29	**27.7**	**29.2**	**29.2**
3DTK	0.00383	10.2	9.32	7.41	3.97	612	780
ALGLIB	1.11	**36100**	**2360**	**415**	**69.6**	152	384
ANN k-d tree	1.16	35.4	-	-	-	**54.6**	**54.6**
ANN BD-tree	0.704	36.1	-	-	-	**44.9**	**54.6**
Boost	2.87	**47900**	**3660**	**560**	**61.1**	79.1	185
CGAL k-d tree	**33.2**	**9690**	**4060**	**625**	**62.9**	77.2	111
FLANN	1.98	78.6	55.5	25.4	7.64	**54.8**	**54.8**
KDTREE1	0.388	99.5	4.9	0.901	0.18	127	292
KDTREE2	2.83	**10400**	**571**	**97.6**	**19.2**	**45.8**	**45.8**
KDTREE3	1.65	**5370**	228	35.5	6.33	59.6	127
KDTREE4	0.297	563	65.3	15.3	3.55	115	115
libkdtree++	0.11	1120	95.8	20.2	4.49	272	272
libkdtree2	0.012	**24000**	**994**	**109**	**12.5**	285	285
libnabo	1.5	69.1	64.5	33	8.63	73.4	94.8
libspatialindex	0.00283	2380	324	44.8	5.56	147	196
nanoflann	0.43	83.5	59	27.4	8.45	**20.7**	**20.7**
nanoflann (duplicated data)	1.09	89	61.5	28.3	8.65	69.4	79.1
Octree	**4.92**	**8180**	**526**	**98.2**	**20.9**	**29.5**	**29.5**
PCL k-d tree	0.708	**5600**	256	40.4	0.965	**51.5**	60.5
PCL octree	**4.61**	140	112	**77.1**	4.35	**51.5**	60.5
PicoTree	2.12	**23500**	**1620**	**311**	**57.5**	105	111
R-tree	0.183	**42300**	**2720**	**447**	**68.9**	123	123
Spatial	0.451	**24000**	330	32.3	3.6	108	212

[1] The coloring scheme is explained in Section IV-B.

The results can be seen in Table III. A few things should be noted about these results. First, as indicated in the table, the libnabo results are for the tree (heap) with $O(\log n)$ query times rather than the $O(n)$ linear vector heap. Second, we use KDTREE2's option to rearrange the data for better performance. We found this offers substantially better query performance with only a slight decrease in build throughput. Third, nanoflann allows you to create an adaptor for an existing vector so that the storage is not duplicated. We show the results for both no duplicated data ("nanoflann") and duplicate data ("nanoflann duplicated data"). Finally, three libraries, KDTREE2, libkdtree2, and Octree, have artificially reduced memory requirements because they only support storing coordinate data as floats rather than doubles (and thus offer lower precision).

From the results we can see that the CGAL k-d tree has by far the highest insertion throughput, followed by Octree and the PCL octree. However, even one of the slower libraries, R-tree, builds in approximately 7 seconds so it should be kept in mind that (apart from 3DTK and libspatialindex) all of these trees build quite quickly. This highlights the performance benefits of being able to parallelize construction (and querying) across the processes through domain decomposition, rather than attempting to use a single index for the entire mesh (although this does increase the total memory consumption). Shockingly, CGAL is able to create the k-d tree significantly faster than the brute force solution copies the 2D vector of mesh coordinates. There are a few libraries that perform well across all query sizes: ALGLIB, Boost, the CGAL k-d tree, KDTREE2, libkdtree2, Octree, PicoTree and R-tree.

Insight: query performance is strongly dictated by the number of matching mesh points and when a large number of points are returned, most libraries fail to beat the brute force solution.

The poor performance on the large queries is a reflection of the fact that all of these libraries are tree based and as such can offer a significant performance advantage when a large portion of the search space can be pruned and a significant performance penalty when large portions of the tree must be traversed. It should also be noted that all of the libraries use far more memory than the theoretical lower bound of linear storage. Almost all of the libraries use between 2-5× the space needed for just the raw coordinate points (29.2 MB), and several use significantly more than this. Only nanoflann with no duplicated storage is able to use anywhere close to linear memory, but it performs more than 10× slower than the best performing library for all query sizes.

2) Small Scale Mesh Elements: Each query of extra-small, small, medium, and large size retrieves an average of 0.00213%, 0.212%, 1.68% and 12.7% of the process's assigned mesh elements.

TABLE IV: Small scale results for mesh elements. We find generally good performance, moderate memory overheads and very good performance compared to brute force[1].

Configuration	Writes/sec (millions)	Queries/sec				Mem. (MB)	Peak Mem. (MB)
		XS	S	M	L		
Brute force	**2.23**	15.9	15.8	15.7	**14.9**	**47**	**47**
Boost	**2.76**	**28000**	**2150**	**267**	**33.8**	**55.9**	154
CGAL R-tree	**15.3**	**4170**	**1420**	**228**	**34.8**	**75.3**	96.3
libspatialindex	0.051	2140	213	32.8	4.64	**72.4**	227
R-tree	0.18	**33000**	**2450**	**417**	**67.4**	102	102
Spatial	0.393	1010	233	40.2	5.78	110	198

[1] The coloring scheme is explained in Section IV-B.

The results of this evaluation are shown in Table IV. One library that we test, CGAL's segment tree, runs into out of memory errors, and is therefore not included in the results. In these results we again find that a few of the libraries' data structures perform well across the board: Boost, the CGAL R-tree, and the R-tree library. In this case, the libraries overall use (relatively) far less memory, with most using less than 2× the memory needed for the raw element coordinate data (47.0MB). Unlike with the point results, here all of the best performing libraries outperform brute force at query sizes (including the large queries) since the brute force solution experiences an approximately 2× slowdown as it now has to consider two points per vector element (the lower and upper corner of the bounding box) rather than just one.

TABLE V: Large scale results for mesh points for libraries that perform best at small scale. Significant differences emerge at large scale[1].

Configuration	Writes/sec (millions)	Queries/sec				Mem. (MB)	Peak Mem. (MB)
		XS	S	M	L		
Brute force	**4.48**	8.08	8.08	8.04	**7.71**	**240**	**240**
ALGLIB	1.03	**29300**	**1270**	**170**	**21.7**	1180	3170
Boost	2.44	**19400**	**939**	**121**	**15**	568	1500
CGAL k-d tree	**29.8**	1660	**1140**	**143**	**17.6**	537	886
KDTREE2	2.12	1610	63.1	12.3	2.65	**374**	**374**
libkdtree2	0.00285	**7720**	**163**	16.1	1.76	2340	2340
Octree	3.28	1440	73.1	14.7	3.21	**234**	**234**
PicoTree	1.53	**16200**	**815**	**132**	**23.5**	852	886
R-tree	0.166	**27000**	**1290**	**175**	**22.5**	1020	1020

[1] The coloring scheme is explained in Section IV-B.

3) Large Scale Mesh Points: Each query of extra-small, small, medium, and large size retrieves an average of 0.00117%, 0.0996%, 1.05% and 9.84% of the process's assigned mesh points. Results are shown in Table V. Although at small scale the libraries have good performance for all query sizes, at large scale we find relatively worse performance for KDTREE2, libkdtree2 and Octree, and for the CGAL k-d tree on extra-small queries. The rest still perform well across the board.

Insight: at large scale the libraries outperform the brute force solution at all query scales. These results can help estimate when the brute force solution offers the best performance.

4) Large Scale Mesh Elements: The queries of extra-small, small, medium, and large size retrieve an average of 0.00222%, 0.120%, 1.17% and 10.5% of the mesh elements assigned to each process. Results are shown in Table VI. Once again, we see that CGAL (this time with an R-tree) struggles with the extra-small queries but has exceptionally fast build throughput, and Boost performs well across the board. Here Boost is the only library that achieves good results in terms of memory requirements since the raw data takes 472 MB. Again, the R-tree library has significantly slower insertion throughput, but offers around a 2× speedup for the medium and large queries.

TABLE VI: Large scale results for hexahedral mesh elements for libraries that perform best at small scale. CGAL offers the best write performance, R-tree generally offers the best read performance, and Boost does well across the board[1].

Configuration	Writes/sec (millions)	Queries/sec				Mem. (MB)	Peak Mem. (MB)
		XS	S	M	L		
Brute force	**2.43**	4.34	4.34	4.3	**3.97**	**473**	**473**
Boost	**2.15**	**15600**	**698**	**78.8**	**8.93**	**637**	1890
CGAL R-tree	**15.3**	424	**530**	**70.5**	**8.87**	1490	1490
R-tree	0.164	**17500**	**976**	**143**	**19.4**	1060	1060

[1] The coloring scheme is explained in Section IV-B.

C. Scalability Evaluations

In this section, we present an evaluation of the strong and weak scaling for the libraries.

1) Strong Scaling: We evaluate the strong scaling by comparing the performance as we use 10× fewer processes (16 instead of 160) for the same mesh. Thus, the data per process increases from small to large (by 8.28× for the point tests and 10.0× for the element tests). The results are shown in Table VII and Table VIII. The values shown are the factor difference (the large result divided by the small result). For the performance results, we classify the scaling as good (green with bolded text) if it is better than logarithmic ($\frac{1}{\log_2 8.28} = 0.328$ for the points and $\frac{1}{\log_2 10.0} = 0.301$ for the elements). We classify the scaling as moderate (yellow with plain text) if it is worse than logarithmic but better than half of logarithmic. All other results are classified as poor (red with underlined text). For the memory results, we classify the result as good (green) if there is a sub-linear increase in memory usage, moderate (yellow) if the increase is greater than linear but less than a 1.5×, and poor (red) if there is a greater than 1.5× increase.

TABLE VII: Strong scaling results for mesh points. Few libraries have good query scalability[1].

Configuration	Writes/sec	Queries/sec				Mem. (MB)	Peak Mem. (MB)
		XS	S	M	L		
Brute force	**0.81**	0.28	0.28	0.28	0.28	**8.22**	**8.22**
ALGLIB	**0.93**	**0.81**	**0.54**	**0.41**	0.31	**7.76**	**8.26**
Boost	**0.85**	**0.41**	0.26	0.22	0.25	**7.18**	**8.11**
CGAL k-d tree	**0.90**	0.17	0.28	0.23	0.28	**6.96**	**7.98**
KDTREE2 rearranged	**0.75**	0.15	0.11	0.13	0.14	**8.16**	**8.16**
libkdtree2	0.24	0.32	0.16	0.15	0.14	**8.21**	**8.21**
Octree	**0.67**	0.18	0.14	0.15	0.15	**7.95**	**7.95**
PicoTree	**0.72**	**0.69**	**0.50**	**0.42**	**0.41**	**8.11**	**7.98**
R-tree	**0.91**	**0.64**	**0.47**	**0.39**	**0.33**	**8.29**	**8.29**

[1] The coloring scheme is explained in Section IV-C1.

a) Mesh Points: As shown in Table VII, apart from libkdtree2, the libraries scale very well in terms of insertion throughput and they almost uniformly have a sub-linear increase in memory usage. However, KDTREE2, libkdtree2 and Octree experience significantly worse than logarithmic scaling for most or all query sizes. Only PicoTree and R-tree experience good scaling across the board, with ALGLIB experiencing good scaling at most query sizes. CGAL's k-d tree experiences moderate scaling for most query sizes.

TABLE VIII: Strong scaling results for hexahedral mesh elements. The libraries generally have good scalability apart from large queries and memory overheads[1].

Configuration	Writes/sec	Queries/sec				Mem. (MB)	Peak Mem. (MB)
		XS	S	M	L		
Brute force	**1.09**	0.27	0.27	0.27	0.27	10.0	10.0
Boost	**0.78**	**0.56**	**0.32**	**0.30**	0.26	11.4	12.3
CGAL R-tree	**1.00**	0.10	**0.37**	**0.31**	0.25	19.8	15.5
R-tree	**0.91**	**0.53**	**0.40**	**0.34**	0.29	10.4	10.4

[1] The coloring scheme is explained in Section IV-C1.

b) Mesh Elements: From Table VIII we can see that all libraries achieve good scaling for insertion throughput and just miss the cutoff for good scaling for the large query size. For the remaining query sizes, all achieve good scaling apart from CGAL's R-tree with the extra-small query size. Interestingly, they all experience a greater than linear increase in memory usage (in contrast to their sub-linear scaling for point storage), and the CGAL R-tree requires almost 20× the memory despite storing only 10× as many elements.

2) Weak Scaling: To evaluate the weak scaling, we use the same setup used for the performance evaluations with one modification: we only perform 1000 extra-small, small, and medium queries and 100 large queries (1/10th as many queries of each category). This is sufficient to allow us to compare differences between the libraries. Where more than the baseline number of processes (160 and 16) are used, several processes write and read the same mesh data. This allows us to ensure that the performance differences are the result of the weak scaling changes rather than changes in the mesh data. For space reasons, we only present the results for a representative subset of the libraries. For each of the four experiment setups, we present scaling results for one library that performed (more or less) well across the board and one that performed moderately well. This demonstrates how a library should be expected to scale depending on what performance category it falls into. We consider two different weak scaling scenarios: when the number of processes per node is held constant and when the number is varied. The results are shown in Tables IX and X. For both scenarios, we compare the results to the performance for the baseline setup used in the performance evaluations (one node, with 160 processes for a small amount of data per process or 16 processes for a large amount). We classify the scaling as good (green with bold text) if the performance is at most 10% worse than the baseline, moderate (yellow with plain text) if it is $(10\% - 25\%]$ worse than the baseline, and poor (red with underlined text) if the performance is over 25% worse. In each of the tables, the results indicate the percent change from the baseline with a negative number indicating the operation takes less time than the baseline and a positive number indicating the operation takes longer than the baseline.

TABLE IX: Weak scaling results for a subset of the libraries using 30 nodes and a fixed number of processes per node. As expected, the libraries all exhibit approximately linear weak scaling[1].

Configuration	**Scale[2]**	**Mesh Data**	**Writes**	**Queries**			
				XS	**S**	**M**	**L**
CGAL k-d tree	**Small**	**Points**	**4.7%**	**-6.4%**	**-1.1%**	**2.0%**	**2.2%**
KDTREE	**Small**	**Points**	**0.4%**	**-4.6%**	**0.7%**	**-0.6%**	**-2.0%**
Boost	**Small**	**Elems**	**-11.0%**	**-2.9%**	**4.8%**	**-2.7%**	**-7.9%**
Spatial	**Small**	**Elems**	**-2.1%**	**-48.2%**	**-2.6%**	**-0.1%**	**-2.2%**
ALGLIB	**Large**	**Points**	**0.5%**	**7.9%**	**3.4%**	**5.5%**	**0.3%**
KDTREE2	**Large**	**Points**	**0.0%**	**1.4%**	**2.7%**	**2.7%**	**2.3%**
Boost	**Large**	**Elems**	11.6%	19.4%	**9.8%**	**8.7%**	**1.5%**
R-tree	**Large**	**Elems**	**4.7%**	**0.9%**	**-6.3%**	**-1.7%**	**-5.5%**

[1] The coloring scheme is explained in Section IV-C2.
[2] In terms of both mesh data per node and processes per node.

a) Fixed Number of Processes per Node: When the number of processes per node is held constant, we would expect to find linear weak scaling (constant performance when the work per process is held constant) since each process performs independent work and does not use shared resources such as the network or file system. To evaluate this, we look at how the performance changes when using 15 or 30 nodes (compared to 1 node). The 15 and 30 node results show very close agreement (within 5%) so we omit the 15 node results for space reasons. The results are shown in Table IX.

Insight: the libraries achieve approximately linear weak scaling in all cases. This indicates our results should hold at any scale if the processes per node and work per process are held constant.

b) Variable Number of Processes per Node: Next, we evaluate weak scaling when a different number of processes per node is used. When a small amount of data per process is stored, we are already maxing out the processes per node (160). We therefore evaluate when 20, 40, 60, and 80 processes per node are used. For space reasons we only present the results for 20 and 80 processes per node. In this case, we use the same number of total processes (160) but vary the number of nodes. For the case when a large amount of data per process is stored, we only use 16 nodes per process (far below the maximum), and therefore keep the number of nodes constant (one) but increase the number of processes per node. For the mesh point case, we present results for when 32 and 80 processes per node are used. For the mesh elements case we present results for when 32 and 64 processes per node are used because with 80 processes we exhaust the node's memory. The results, shown in Table X, are much more varied. The libraries generally scale well when the number of processes per node is only 2× different from baseline. However, when the number of processes per node is 4, 5 or 8× different the results are sometimes greatly affected. This can be seen when using 64 or 80 processes per node with a large amount of data per process and with 20 processes per node with a small amount of data per process. For example, with a small number of points per process, the CGAL k-d tree (for points) and Boost library (for elements) both perform dramatically better when only 20 processes per node are used. With a large amount of data per process, we can see that ALGLIB and Boost both perform substantially worse when the number of processes per node is increased. Thus, some of the libraries are much more sensitive to the number of processes per node that are being utilized.

TABLE X: Weak scaling results for a representative subset of the libraries using a variable number of processes per node. Some libraries are more sensitive than others to the number of processes per node[1].

Configuration	**Nodes**	**Procs**	**Writes**	**Queries**			
				XS	**S**	**M**	**L**
CGAL k-d tree[2]	**2**	**160**	21.7%	**1.3%**	**6.6%**	**3.8%**	**-16.4%**
CGAL k-d tree[2]	**8**	**160**	38.9%	**-48.5%**	**-19.7%**	**-28.5%**	**-79.2%**
KDTREE[2]	**2**	**160**	**7.7%**	**0.4%**	**1.6%**	**-0.3%**	**-1.9%**
KDTREE[2]	**8**	**160**	**4.7%**	**-5.9%**	**-4.5%**	**-5.7%**	**-6.7%**
Boost[3]	**2**	**160**	**9.1%**	15.4%	33.7%	19.6%	11.6%
Boost[3]	**8**	**160**	**-17.5%**	**-55.1%**	**-49.4%**	**-67.0%**	**-43.5%**
Spatial[3]	**2**	**160**	12.5%	**1.0%**	**0.9%**	**-0.7%**	**-2.7%**
Spatial[3]	**8**	**160**	**-12.2%**	**-56.9%**	**-19.6%**	**-18.0%**	**-20.5%**
ALGLIB[4]	**1**	**32**	**-0.9%**	**-1.9%**	**1.3%**	**-4.4%**	**-14.7%**
ALGLIB[4]	**1**	**80**	17.9%	26.3%	22.2%	16.0%	**2.5%**
KDTREE2[4]	**1**	**32**	**-2.7%**	**-2.4%**	**0.1%**	**-0.2%**	**-6.8%**
KDTREE2[4]	**1**	**80**	**4.9%**	**0.8%**	**6.2%**	**6.2%**	**-2.5%**
Boost[5]	**1**	**32**	14.7%	20.3%	**6.1%**	**-0.5%**	**-6.5%**
Boost[5]	**1**	**64**	25.8%	40.7%	24.0%	14.1%	**-0.9%**
R-tree[5]	**1**	**32**	**7.1%**	**-7.7%**	**-18.4%**	**-13.4%**	**-19.5%**
R-tree[5]	**1**	**64**	**7.2%**	10.8%	**5.8%**	12.6%	**5.4%**

[1] An explanation of the coloring scheme can be found in Section IV-C2.
[2] Mesh points, small # per process
[3] Mesh elems, small # per process
[4] Mesh points, large # per process
[5] Mesh elems, large # per process

D. Applicability of Results

Although we only use a single mesh type in this evaluation (unstructured, hexahedral), we can use these results to estimate the performance we would get with other mesh types. As shown in Tables III through VI, query performance is greatly affected by the query size (percent of stored mesh points or elements that match the query), exhibiting an approximately linear relationship. Therefore, any mesh with more densely clustered points or elements near the range being queried should result in decreased performance. Another factor that should affect performance is the shape of the mesh elements. All of the libraries that support mesh elements (2D or 3D shapes) use axis aligned bounding boxes (AABBs) to determine potential intersections. Thus, the more a mesh element deviates from an AABB, the worse the performance because of the increase in tree branches that must be explored and potential matches identified, and the need to use more complex intersection tests to achieve accurate results. Beyond these factors, we would not expect much of a performance difference. Although many of the libraries use data structures that are not necessarily balanced (e.g., octrees or k-d trees), almost all of the indices are balanced. This is accomplished through a combination of bulk loading and re-balance functions. Therefore, we would not expect to find tree-height related performance differences, or edge cases producing dramatically unbalanced trees. We leave as future work directly evaluating how the use of different mesh types affects these results.

Insight: our query results can be used to extrapolate expected performance for different mesh types.

V. Discussion, Insights and Future Work

In this work, we set out to answer the following questions regarding the 20 free, open-source C/C++ libraries that support range queries:

1) Which of the implementations are viable in HPC?
2) How do these libraries compare in terms of build time, query time, and memory usage at different scales?
3) Is there a single overall best solution?
4) When should a brute force solution be used?

The second question has been thoroughly addressed in the evaluation section (see Section IV). In this section we summarize answers to the remaining questions using the evaluation results and offer a number of insights that can assist both HPC application scientists and spatial index developers. We also present areas for future work.

A. *Which of the Implementations Are Viable in an HPC Setting?*

As mentioned above, to be viable in an HPC setting, a spatial indexing library must have fast performance for building the index and performing queries, good scalability, and moderate memory usage. In this section, we discuss which libraries best meet these criteria.

Overall, the results show that for point storage, ALGLIB, Boost, CGAL's k-d tree, PicoTree and R-tree achieve the best query results. KDTREE2, libkdtree2 and Octree perform well at small scale but achieve worse performance at large scale. Of these five best performing libraries, R-tree experiences orders of magnitude worse insertion throughput. Therefore, if the application will be performing a small number of queries (meaning index construction is a larger proportion of work), the R-tree library should not be used, and CGAL's k-dtree, which offers almost an order of magnitude improvement over the next best library, should be strongly considered. In addition, both the R-tree library and ALGLIB require around 4× more memory than is required for the raw data points. Therefore, these are not viable solutions for particularly memory constrained environments. However, for point storage the R-tree library does offer some of the fastest query throughputs. In fact, if many medium and large queries will be performed at large scale the R-tree library offers the fastest solution providing nearly double the query throughput for medium and large queries.

For storing mesh elements, Boost, CGAL's R-tree and the R-tree library offer the best performance. CGAL's R-tree offers close to a 10× improvement in insertion throughput over Boost and an 100× improvement over R-tree both at large and small scales, but generally offers the worst performance of the three for all query sizes (and particularly poor performance for extra-small queries). It is therefore a viable solution if a relatively small number of queries will be performed. Boost has by far the smallest memory usage (although it has the highest peak memory usage) and is thus the best solution for moderately memory constrained environments that can tolerate temporary memory pressure. Finally, the R-tree library almost always offers the best query performance, performing around 2× better than both Boost and CGAL for the medium and large queries at both small scale and large scale. It is therefore a viable solution when slightly higher memory usage can be tolerated, and a large number of queries will be performed.

It is worth emphasizing that all of the best performing libraries use significantly more memory than the raw data size and therefore are not suitable for severely memory constrained environments. If memory is severely constrained, we would recommend nanoflann with a data adapter (which is simple to write), which results in sub-linear memory usage, or the Octree library for point storage and the Boost library for element storage. For severely space constrained environments, there is another simple option: store floats rather than doubles. This will cut memory requirements approximately in half. We expect that four bytes worth of precision will be sufficient for range queries in most cases given that the spatial index is not taking the place of the mesh coordinate storage, but rather is offering an additional data structure on top (users therefore are not reducing mesh precision just query precision). It should also be kept in mind that it is the number of mesh points per process that determines the total structure size rather than the dataset size. Most simulations store a large number of variables (anywhere from 5-100) and a large number of timesteps (anywhere from 100-100,000) and the number of mesh coordinates or elements will therefore be a relatively small fraction of the overall dataset. It is also worth emphasizing that, given how quickly these structures can be created, they can easily be generated on the fly and therefore will not require any long-term storage.

It is worth noting that the five best performing libraries for storing points use either a k-d tree (ALGLIB, CGAL's k-d tree, PicoTree) or an R-tree (Boost, the R-tree library). For element storage, all of the best performing libraries use an R-tree (Boost, CGAL's R-tree and the R-tree library). Both k-d trees and R-trees can answer a range query in $O(n^{1-1/d} + k)$ time [18] whereas octrees can answer range queries in $O(\log n)$ time [21]. Therefore, it is perhaps a bit surprising that none of the best performing implementations are octrees. Given the wide range of performance we see across the libraries for each tree type, we should expect the library's implementation to be a much greater factor in determining performance than the theoretical bounds of the data structure.

B. *Is There a Single Overall Best Solution?*

As the evaluation section makes clear, there is no universally good library that achieves fast build throughput, fast query throughput for all query sizes and uses close to linear memory. However, Boost, CGAL and R-tree offer some of the best performance at large and small scales and have the advantage of being able to support both points and boxes. Of the three libraries, Boost offers the best overall performance in terms of good build and query times (and has the lowest memory usage), while CGAL offers the best build times and R-tree offers the best query times. There is therefore no single best solution, but rather the answer will depend on the problem scale, memory availability, the number of queries to be performed and the fraction of stored data that is expected to match the queries. In addition, as discussed in the next section, there are circumstances under which the best library to use is no library.

C. When should a brute force solution be used?

The results demonstrate that if many of the queries will retrieve 10% or more of the stored mesh data, and if there is a small amount of data per process (e.g., less than 10 million points or elements), then one should strongly consider using a brute force solution. The libraries evaluated in this paper offer the best performance when the search space can be substantially reduced (pruning branches) and pay a large performance penalty for the $\log n$ tree traversals when a large portion of the tree must be searched. In addition, as discussed previously, all of the libraries require significantly more memory than the raw data requires, and therefore the brute force solution should be used if users are severely memory constrained.

D. Future Work

We leave as future work evaluating these libraries for dynamic use (e.g., if adaptive mesh refinement is used) and for mesh elements that are more complex shapes. We also plan to assess the portability and software design choices of the libraries, and to evaluate how the use of different architectures affects the evaluation results. Finally, we plan to evaluate libraries for languages other than C and C++.

Acknowledgements

This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

This work was supported by the U.S. Department of Energy Office of Science, under the SSIO grant series and the Data Management grant series, Decaf project, program manager Lucy Nowell.

Margaret Lawson acknowledges support from the United States Department of Energy through the Computational Sciences Graduate Fellowship (DOE CSGF) under grant number: DE-SC0020347.

This work was supported in part by the State of Illinois.

References

[1] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu, "Six degrees of scientific data: reading patterns for extreme scale science IO," in *Proceedings of the 20th international symposium on High performance distributed computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/1996130.1996139

[2] S. Papadomanolakis, A. Ailamaki, J. C. Lopez, T. Tu, D. R. O'Hallaron, and G. Heber, "Efficient query processing on unstructured tetrahedral meshes," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2006, pp. 551–562.

[3] H. Hinterberger, K. A. Meier, and H. Gilgen, "Spatial data reallocation based on multidimensional range queries. a contribution to data management for the earth sciences," in *Seventh International Working Conference on Scientific and Statistical Database Management*. New York, NY, USA: IEEE, 1994, pp. 228–239.

[4] H. Lu, S. K. Seal, W. Guo, and J. Poplawsky, "Spherical region queries on multicore architectures," in *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*. New York, NY, USA: ACM, 2017, pp. 1–4.

[5] S. Papadomanolakis, A. Ailamaki, J. C. Lopez, T. Tu, D. R. O'Hallaron, and G. Heber, "Efficient query processing on unstructured tetrahedral meshes," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2006, pp. 551–562.

[6] J. Beyer, A. Al-Awami, N. Kasthuri, J. W. Lichtman, H. Pfister, and M. Hadwiger, "Connectomeexplorer: Query-guided visual analysis of large volumetric neuroscience data," *IEEE transactions on visualization and computer graphics*, vol. 19, no. 12, pp. 2868–2877, 2013.

[7] M. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz, "Datacutter: Middleware for filtering very large scientific datasets on archival storage systems," in *IEEE Symposium on Mass Storage Systems*. New York, NY, USA: IEEE, 2000, pp. 119–134.

[8] M. F. Barone, J. Ray, and S. Domino, "Feature selection, clustering, and prototype placement for turbulence data sets," in *AIAA Scitech 2021 Forum*. Reston, VA: American Institute of Aeronautics and Astronautic, 2021. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2021-1750

[9] L. Gosink, J. Anderson, W. Bethel, and K. Joy, "Variable interactions in query-driven visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1400–1407, 2007.

[10] M. Betoule, J. Marriner, N. Regnault, J.-C. Cuillandre, P. Astier, J. Guy, C. Balland, P. El Hage, D. Hardin, R. Kessler *et al.*, "Improved photometric calibration of the snls and the sdss supernova surveys," *Astronomy & Astrophysics*, vol. 552, p. A124, 2013.

[11] A. Aji, F. Wang, and J. H. Saltz, "Towards building a high performance spatial query system for large scale medical imaging data," in *Proceedings of the 20th international conference on advances in geographic information systems*. New York, NY, USA: ACM, 2012, pp. 309–318.

[12] H.-T. Chiu, J. Chou, V. Vishwanath, and K. Wu, "In-memory query system for scientific datasets," in *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. New York, NY, USA: IEEE, 2015, pp. 362–371.

[13] M. Rasquin, C. Smith, K. Chitale, S. Seol, B. Matthews, J. Martin, O. Sahni, R. Loy, M. S. Shephard, and K. E. Jansen, "Scalable fully implicit finite element flow solver with application to high-fidelity flow control simulations on a realistic wing design," *Computing in Science and Engineering*, vol. 16, no. 6, pp. 13–21, 2014.

[14] Z. Zhao, Y. Zhang, L. He, X. Chang, and L. Zhang, "A large-scale parallel hybrid grid generation technique for realistic complex geometry," *International Journal for Numerical Methods in Fluids*, vol. 92, no. 10, pp. 1235–1255, 2020.

[15] C. Godenschwager, F. Schornbaum, M. Bauer, H. Köstler, and U. Rüde, "A framework for hybrid parallel flow simulations with a trillion cells in complex geometries," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[16] U. D. of Energy. (2019) U.s. department of energy and intel to build first exascale supercomputer. U.S. Department of Energy. [Online]. Available: https://www.energy.gov/articles/us-department-energy-and-intel-build-first-exascale-supercomputer

[17] J. Matoušek, "Geometric range searching," *ACM Computing Surveys (CSUR)*, vol. 26, no. 4, pp. 422–461, 1994.

[18] P. K. Agarwal, "Range searching," DUKE UNIV DURHAM NC DEPT OF COMPUTER SCIENCE, Tech. Rep., 1996.

[19] D. E. Willard, "New data structures for orthogonal range queries," *SIAM Journal on Computing*, vol. 14, no. 1, pp. 232–253, 1985.

[20] B. Chazelle, "Lower bounds for orthogonal range searching: I. the reporting case," *Journal of the ACM (JACM)*, vol. 37, no. 2, pp. 200–212, 1990.

[21] C. L. Jackins and S. L. Tanimoto, "Oct-trees and their use in representing three-dimensional objects," *Computer Graphics and Image Processing*, vol. 14, no. 3, pp. 249–270, 1980.

[22] L. Formaggia, "Data structures for unstructured mesh generation," 1999.

[23] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[24] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, 1984, pp. 47–57.

[25] S. T. Leutenegger, M. A. Lopez, and J. Edgington, "Str: A simple and efficient algorithm for r-tree packing," in *Proceedings 13th International Conference on Data Engineering*. New York, NY, USA: IEEE, 1997, pp. 497–506.

[26] Y. J. García R, M. A. López, and S. T. Leutenegger, "A greedy algorithm for bulk loading r-trees," in *Proceedings of the 6th ACM international symposium on Advances in geographic information systems*, 1998, pp. 163–164.

[27] J. Elseberg, S. Magnenat, R. Siegwart, and A. Nüchter, "Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration," *Journal of Software Engineering for Robotics*, vol. 3, no. 1, pp. 2–12, 2012.

[28] V. Amaral, B. Norberto, M. Goulão, M. Aldinucci, S. Benkner, A. Bracciali, P. Carreira, E. Celms, L. Correia, C. Grelck *et al.*, "Programming languages for data-intensive hpc applications: A systematic mapping study," *Parallel Computing*, vol. 91, p. 102584, 2020.

[29] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, "A large-scale study of mpi usage in open-source hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: ACM, 2019, pp. 1–14.

[30] K. Mehlhorn and S. Näher, *LEDA: A platform for combinatorial and geometric computing*. Cambridge university press, 1999.

[31] A. Nüchter and K. Lingemann. (2011) 3dtk–the 3d toolkit. [Online]. Available: http://slam6d.sourceforge.net

[32] S. Bochkanov and V. Bystritsky, "Alglib-a cross-platform numerical analysis and data processing library," *ALGLIB Project. Novgorod, Russia*, 2011.

[33] S. Arya and D. Mount, "Ann: library for approximate nearest neighbor searching," in *Proceedings of IEEE CGC Workshop on Computational Geometry, Providence, RI*, 1998.

[34] B. Gehrels, B. Lalande, M. Loskot, A. Wulkiewicz, M. Karavelas, and V. Fisikopoulos. (2020) Geometry. Boost. [Online]. Available: https://www.boost.org/doc/libs/1_75_0/libs/geometry/doc/html/index.html

[35] A. Fabri and S. Pion, "Cgal: The computational geometry algorithms library," in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. GIS '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 538–539. [Online]. Available: https://doi.org/10.1145/1653771.1653865

[36] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration." *VISAPP (1)*, vol. 2, no. 331-340, p. 2, 2009.

[37] J. F. Carvalho. (2020) Kdtree. [Online]. Available: https://github.com/crvs/KDTree

[38] M. B. Kennel, "Kdtree 2: Fortran 95 and c++ software to efficiently search for near neighbors in a multi-dimensional euclidean space," *arXiv:physics/0408067*, 2004.

[39] Anon. (2020) Kdtree. [Online]. Available: https://github.com/G3tupup/KdTree

[40] J. Tsiombikas. (2019) kdtree. [Online]. Available: https://github.com/jtsiomb/kdtree

[41] M. F. Krafft. (2020) libkdtree. [Online]. Available: https://github.com/nvmd/libkdtree

[42] J. Dietrich. (2013) libkdtree. [Online]. Available: https://github.com/joergdietrich/libkdtree

[43] M. Hadjieleftheriou, "Libspatialindex," 2015.

[44] J. L. Blanco and P. K. Rai. (2014) nanoflann: a c++ header-only fork of flann, a library for nearest neighbor (nn) wih kd-trees. [Online]. Available: https://github.com/jlblancoc/nanoflann

[45] J. Behley, V. Steinhage, and A. B. Cremers, "Efficient radius neighbor seach in three-dimensional point clouds," in *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[46] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *2011 IEEE international conference on robotics and automation*. New York, NY, USA: IEEE, 2011, pp. 1–4.

[47] J. Broere. (2021) Picotree. [Online]. Available: https://github.com/Jaybro/pico_tree

[48] M. Green and G. Douglas. (2004) R-trees. [Online]. Available: https://superliminal.com/sources/#C_Code

[49] S. Bougerel. (2017) Spatial c++ library. [Online]. Available: http://spatial.sourceforge.net/index.html

[50] M. Lawson. (2021) Benchmarking suite for range tree libraries. [Online]. Available: https://github.com/mlawsonca/benchmarking_suite_range_searching_libraries

[51] S. P. Domino, P. Sakievich, and M. Barone, "An assessment of atypical mesh topologies for low-mach large-eddy simulation," *Computers & Fluids*, vol. 179, pp. 655–669, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045793018305942

[52] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp, "Modeling the performance of an algebraic multigrid cycle on hpc platforms," in *Proceedings of the international conference on Supercomputing*. New York, NY, USA: ACM, 2011, pp. 172–181.

[53] M.-C. Hsu, I. Akkerman, and Y. Bazilevs, "High-performance computing of wind turbine aerodynamics using isogeometric analysis," *Computers & Fluids*, vol. 49, no. 1, pp. 93–100, 2011.

[54] A. Quintanas-Corominas, P. Maimí, E. Casoni, A. Turon, J. A. Mayugo, G. Guillamet, and M. Vázquez, "A 3d transversally isotropic constitutive model for advanced composites implemented in a high performance computing code," *European Journal of Mechanics-A/Solids*, vol. 71, pp. 278–291, 2018.

[55] E. Gabriel, S. Feki, K. Benkert, and M. M. Resch, "Towards performance portability through runtime adaptation for high-performance computing applications," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 16, pp. 2230–2246, 2010.

[56] T. Marrinan, G. Eisenhauer, M. Wolf, J. A. Insley, S. Rizzi, and M. E. Papka, "Parallel streaming between heterogeneous hpc resources for real-time analysis," *Journal of Computational Science*, vol. 31, pp. 163–171, 2019.

[57] C. Peña-Monferrer, R. Manson-Sawko, and V. Elisseev, "Hpc-cloud native framework for concurrent simulation, analysis and visualization of cfd workflows," *Future Generation Computer Systems*, vol. 123, pp. 14–23, 2021.

[58] J. Cummings, A. Pankin, N. Podhosrzki, G. Park, S. Ku, R. Barreto, S. Klasky, C. Chang, H. Strauss, L. Sugiyama *et al.*, "Plasma edge kinetic-mhd modeling in tokamaks using kepler workflow for code coupling, data management and visualization," *Communications in Computational Physics*, vol. 4, no. 3, pp. 675–702, 2008.

[59] N. A. Simakov, J. P. White, R. L. DeLeon, S. M. Gallo, M. D. Jones, J. T. Palmer, B. Plessinger, and T. R. Furlani, "A workload analysis of nsf's innovative hpc resources using xdmod," *arXiv preprint arXiv:1801.04306*, 2018.

[60] M. A. Salim, T. D. Uram, J. T. Childers, P. Balaprakash, V. Vishwanath, and M. E. Papka, "Balsam: Automated scheduling and execution of dynamic, data-intensive hpc workflows," *arXiv preprint arXiv:1909.08704*, 2019.

[61] (2020) Massif: a heap profiler. [Online]. Available: https://www.valgrind.org/docs/manual/ms-manual.html

[62] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.